Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments

Jérémy Dubus, Philippe Merle

Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments

Jérémy Dubus[1] and Philippe Merle[2]

[1] **Laboratoire d'Informatique Fondamentale de Lille - UMR CNRS 8022**
GOAL / INRIA ADAM Team
Université des Sciences et Technologies de Lille - Cité Scientifique
59655 Villeneuve d'Ascq CEDEX— FRANCE
Email: Jeremy.Dubus@lifl.fr
[2] **INRIA - ADAM Team**
INRIA FUTURS Parc Scientifique de la Haute Borne
Avenue Halley B.P. 70478
59658 Villeneuve d'Ascq CEDEX — FRANCE
Email : Philippe.Merle@inria.fr

Abstract. New distributed systems are running onto fluctuating environments (*e.g.* ambient or grid computing). These fluctuations must be taken into account when deploying these systems. *Autonomic computing* aims at realizing programs that implement self-adaptation behaviour. Unfortunately in practice, these programs are not often statically validated, and their execution can lead to emergent undesirable behaviour. In this paper, we argue that static validation is mandatory for large autonomic distributed systems. We identify two kinds of validation that are relevant and crucial when deploying such systems. These validations affect the deployment procedures of software composing a system, as well as the autonomic policies of this system. Using our DACAR model-based framework for deploying autonomic software distributed architectures, we show how we tackle the problem of static validation of autonomic distributed systems.

1 Introduction

The nature of the networks used to deploy distributed systems is changing. Well-defined networks, with well-known hosts, are no longer employed. The new emerging environments are *Open Dynamic Distributed Environments* (ODDE). Ambient, grid or sensor networks are the most known of these ODDE. In such environments, hosts can appear or disappear at any time. These changes in the environment have an impact on the applications deployed, hence these fluctuations must be taken into account to adapt properly their software architectures.

Autonomic computing [1] proposes to solve the problem of software self-adaptation, by introducing the concept of *autonomic policies*, which are the entities in charge of ensuring the adequate runtime reconfiguration of the system. The whole set of autonomic policies defined for a system represents what we call the *autonomic behaviour* of this system. Unfortunately, all environments that have emerged from this paradigm propose to write autonomic policies in a programmatic way, or using reconfiguration scripts. Programs or scripts that implement autonomic behaviour of a system only enable syntactical or semantical verifications. But we argue that these verifications are not sufficient. Indeed our opinion is that behaviour verifications are also needed to ensure that the system will not reach unconsistent behaviour state in some cases. We have identified two kinds of validations that support our position.

First, long-life systems that are modified very often during execution: software are installed then uninstalled, and this is repeated several times. In such a context, we have to ensure that every action performed in some installation process is undone in the opposite uninstallation process. Otherwise, undesirable side effects (*e.g.* a process started in the installation procedure that is not killed in the uninstallation procedure)

can occur and grow during the lifecycle of the system. These side effects can somehow lead to a crash of the system.

Second, autonomic policies can also interfere with each other and produce unexpected emergent behaviour (*e.g.* infinite loops). The problem has already been partially identified in the domain of active databases and is known as the *Feature Interaction* problem [2].

In this paper, we also introduce our proposition to solve the issue of validation of large and complete software autonomic systems in ODDE. This proposition relies on our DACAR model-based framework for building autonomous architectures [3, 4]. In this paper, we propose to extend the DACAR metamodel in order to validate autonomic policies. Two validations must be performed : First every action performed when a software is installed/started must be cancelled when this software is uninstalled/stopped. Second, the autonomic policies must be introspected to detect feature interactions such as cycles possible in the policy stack execution.

The remainder of this paper is the following. Section 2 presents the key research challenges of this work. Section 3 explains how we extend our DACAR metamodel in order to handle the issue of ODDE autonomic systems validation. In Section 4, we expose the research work related to our proposition. Finally, we discuss our future work and conclude in Section 5.

2 Key Research Challenges

Deployment of complex software systems has become a nightmare for administrators. This deployment procedure essentially consists in accomplishing tasks to set up all middleware servers, as well as to deploy every business component upon these servers. In a fluctuating environment such as ODDE, it is impossible to manually perform these tasks, since the target machines hosting the applications are unknown. Moreover after the initial deployment, new machines can appear or disappear and manual administration intervention is needed. From this statement emerged the Autonomic Computing paradigm, which consists in extending programs with self-adaptation mechanisms. The core principles of the autonomic computing rely on the *control loop*, represented on Figure 1.

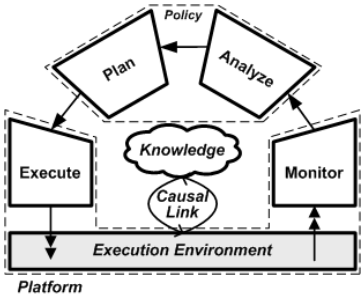

Fig. 1. The control loop of autonomic computing

This loop consists in four phases: Monitoring the system, analyzing the situation and taking a decision about some monitored changes, planning the adequate reconfiguration actions, and executing them. The analyzing and planning phases are relying on the *Knowledge* part as support for computation. It is an abstract representation of the autonomous system. At runtime the Knowledge part must always be conform to the execution environment, which means that every change in the execution environment must lead to an update of the Knowledge part, and *vice-versa*. Then a causal link must be maintained between the Knowledge part and the execution environment. Nevertheless, with the hypothesis of designing and deploying such autonomic systems are raised some issues, that are exposed in the following subsections.

2.1 Expression of Autonomic Behaviour

The first challenge to face considers the expression of autonomic behaviour of software systems. Adequate concepts must be identified for the administrators to express precisely and naturally their *autonomic policies*, *i.e.* the behaviour that they want to inject into their software architectures. Secondly, it is important that the paradigm for expressing software system behaviour at runtime is independent from any technologies. The mechanism that executes this behaviour must also be generic in order to apply this approach to software designed using any of these technologies. Finally, these concepts must be independent from the granularity of the software entity. Indeed, considering the deployment of whole software systems, administrators have to handle both with fine granularity business components, as well as with coarse granularity middleware servers. The autonomic behaviour paradigm must allow the administrator to write its autonomic policies for both of them.

2.2 Unit Deployment Validation

The first step to build an autonomic deployment process in an ODDE consists in writing procedures to install, configure and start pieces of software, and also procedures to stop, unconfigure and uninstall them. Autonomic mechanisms will then call these procedures at runtime according to the changes of the environment with respect to the global policy. Therefore, the first required validation concerns these procedures. Validating these procedures means ensuring that every instruction in a procedure (*e.g.* install, configure, start) must be cancelled in the opposite procedure (*e.g* uninstall, unconfigure, stop). This first step in validation then allows the administrators to write their autonomic policies using validated and safe deployment procedures for the different software involved in the system.

Here is a concrete simple example of such a problem. Considering a CORBA component-based application that has to be extended to every mobile phone entering the domain. On each mobile phone, a CORBA component server must be deployed, and then started (let's assume that this start procedure consists in launching a daemon). When this mobile phone leaves, a local autonomic policy must undeploy the component server and then launch the uninstallation procedure, which consists in removing the directory in which this component server was downloaded from the local filesystem. This action, of course, does not kill the component server daemon. Let's now consider that this mobile phone joins the domain again, and repeats this sequence (leave the domain, join the domain) several times: The CORBA component server will be started again several times on the mobile phone, and this can leads to a memory overflow, in that mobile phone.

2.3 Validation of Autonomic Policies

The last challenge concerns the autonomic policies themselves. These policies, according to the control loop, rely on the following principle: Then a *stimulus* occurs, under some *contextual conditions*, apply the adequate *reconfiguration actions*. However policies written using this paradigm can interfere with each other, as shown in [2], and there are many kinds of feature interactions. For instance, two different policies can be triggered by one unique stimulus, this is called the *Shared Trigger Interaction*. Another important feature interaction is the *Looping Interaction*: The reconfiguration action of a policy P1 can lead to the trigger of another policy P2 whose reconfiguration action leads to another succession of policy triggers that finally triggers the P1 policy. We fall into a cycle in policy execution. The list of feature interactions given here is not exhaustive. Using a strictly programmatic way to implement autonomic policies, it is impossible to detect such interaction between rules. So the challenge is to provide concepts that enable validation of policies.

Here is a concrete example of such a problem (represented on Figure 2). Three component types are involved in this example : `ClientComp` which has two required ports (logC and servC), `ServerComp` which has one provided port (servS) and `LogComp` which has one provided port (logS). We suppose that we have four autonomic policies expressed in an unformal paradigm:

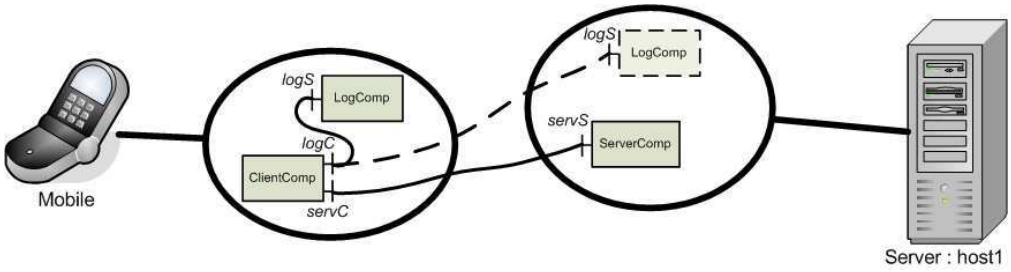

Fig. 2. An example involving cycling autonomic policies

RULE1 When a PDA enters the network, the deployment of a CORBA component server is launched and the deployment of a ClientComp component is performed on top of it.

RULE2 When a `ClientComp` (CC) is deployed onto a PDA, a remote binding is established between `CC.servC` and `SERVER.servS` (`SERVER` is a statically known instance deployed on the Host *host1*.)

RULE3 When a remote binding is made from a ClientComp instance (CC) required port, then a `LogComp` instance (LC) is deployed onto the PDA's component server (in order to log communications made through this binding), and a binding between `CC.logC` and `LC.logS` is established.

RULE4 When a remote binding is made from a `ClientComp` instance (CC) required port BUT there is no memory available to deploy the LogComp instance, then the `LogComp` instance (LC) is deployed on the Host *host1* and a binding between `CC.logC` and `LC.logS` is established.

One by one, these rules seem to be coherent and relevant to the application. Nevertheless a cycle can occur in the application of these rules: If the RULE4 is triggered, the binding between LC and CC becomes a remote binding, then the RULE4 is called again : the global autonomic behaviour falls into a cycle (for the sake of simplicity, the cycle here is very simple: a policy infinitely calls itself).

3 Our DACAR proposition

In this section we introduce some general details about our DACAR approach to execute autonomic system deployment in ODDE. DACAR allows to deploy complete software systems, from low-level installation of middleware servers to deployment of fine-grained business components.

3.1 Principles of the DACAR Approach

DACAR is based on a control loop, where the Knowledge model is implemented using models that abstract any relevant information about the underlying system. As can be seen on Figure 3, each part of the causal link between Knowledge and execution environment is implemented with two kinds of rules: The *Monitoring* and the *Deployment Rules*. The first one consists in monitoring the execution environment, and in case of change emerging from there, to reconfigure the Knowledge model. The second one consists in observing changes emerging from the Knowledge model and to apply these changes on the execution environment. The autonomic behaviour of the system is implemented through *Architectural Rules*. These rules represent the autonomic policies in our approach. As we have investigated in [4], models represent an adequate support to express all relevant information about the system, so fits well in the role of the knowledge model. We also identified that Event-Condition-Action (ECA) is a well tailored paradigm to express the causality between knowledge and execution environment as well as to express the autonomic behaviour.

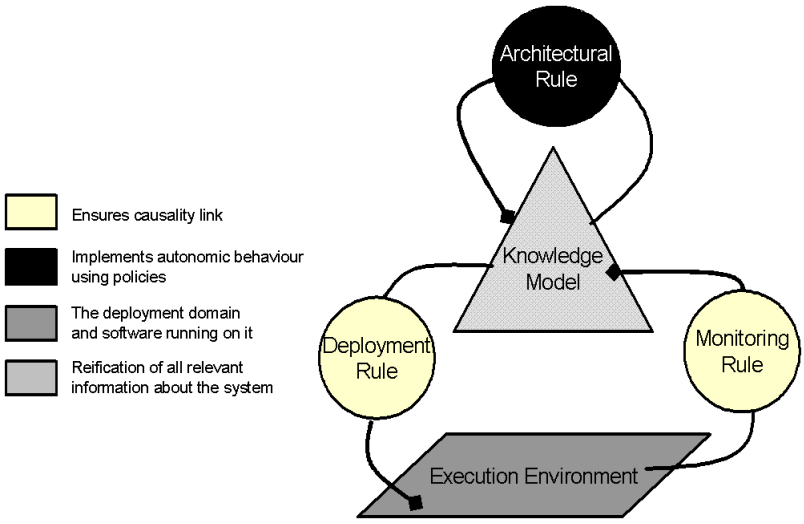

Fig. 3. Overview of the DACAR approach

3.2 Structure of the DACAR Metamodel

The first DACAR prototype presented a proof of feasibility of the approach, showing how to add autonomic behaviour in component-based software architectures (as it was CCM components). To achieve that, we have proposed the OMG D&C specification [5] as the Knowledge metamodel, but we found out that this specification only considers business components and does not allow multi-granularity deployment modeling. Moreover this specification encompasses several complex concepts that are not all useful for describing autonomic system deployment. In DACAR, it is possible to seamlessly express deployment and configuration of middleware servers as well as fine-grained business components, thereby a generic metamodel which exactly fits our needs has been established. Therefore our motivation is then to have a metamodel that focuses precisely on the deployment and autonomic concerns of a system, independently from the granularity chosen. This metamodel, represented on Figure 4 contains three subpart : the first one to define the deployment procedures of software systems, the second one to define the autonomic policies woven onto these systems, and the third only defines validation-specific concepts.

This metamodel expresses the main concepts about the deployment of several *Software* that are connected together to form a *System*, on low-left part of Figure 4. The deployment of a *Software* can depend on the deployment of other Software (*e.g.* a Java EE server depends on the Java runtime). Consequently, the sequence of deployment procedure will be scheduled according to these dependencies among Software. A Software is defined by several *Properties* such as the archive to download, where to install it, and other specific properties of the software. *Procedures* are also contained in a Software in order to install, maybe configure and start the Software as well as stop, uninstall the software or any other procedure specific to the Software deployed. These Procedures are composed of primitive *Instructions* to set environment variables, execute processes, etc. The *Host* is also represented in this metamodel in order to specify important access information about a specific target machine, such as file transfer (*e.g.* FTP) or remote access protocol (*e.g.* SSH). This Software metaclass is based on our another work which is called DEPLOYWARE. This work focuses on the execution of the deployment of a whole Software system according to a high-level description, and with respect to the dependencies of the system. This remains an ongoing work which is introduced in [6].

The first extension to this initial metamodel represents the answer to the first challenge announced in Section 2 and is about the expressivity of autonomic policies. Here we expose the autonomic part of our metamodel that allows the administrators to seamlessly express the autonomic policies of their systems

using ECA-like policies. This subpart of our metamodel is represented on the low-right part of Figure 4. A global autonomic behaviour of a *Software* element consists in a set of *Autonomic Policies*. A Policy is defined by an *Event* which is the concept that reifies a change occuring during the execution of the system. Under some *Conditions*, that depend on any element of the model, a set of *Actions* is triggered. This concept of action reifies any modification of the current model, which encompasses creation of a new software entity, reconfiguration of a property of a Software, or calling some Procedure of a Software. This metamodel is independent from any technology but also from any software granularity.

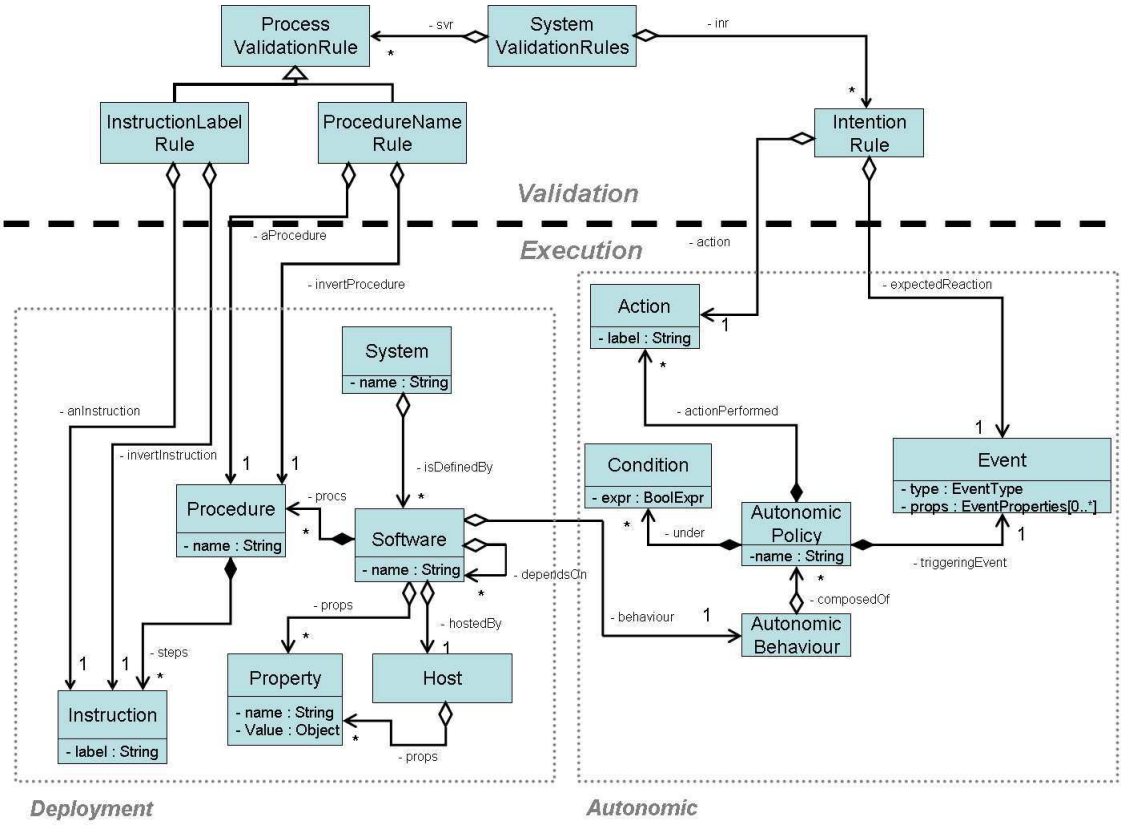

Fig. 4. The DACAR metamodel

3.3 Introducing Concepts in the Metamodel for Validation

In this section we expose the other concepts of the DACAR metamodel to answer the remaining challenges raised in Section 2. These validation-specific concepts are represented in the upper part of Figure 4.

Unit Deployment Validation — To face this challenge, we introduce the concept of *Process Validation Rule* (PVR). This particular type of rules is divided in two categories: The *Procedure Name Rules* (PNR) and the *Instruction Label Rules* (ILR). The first category allows the administrator to associate Procedures that have opposite goals. An example of PNR is the association between a `start` procedure and a `stop` procedure. Introducing such a rule ensures that, in a Software encompassing this Rule if the `start` procedure

is present, its invert stop procedure must also exist in the software. The ILR allows a more fine-grained verification into the procedures to inspect the instructions. This is an association between two opposite instructions. For example the instruction startProcess(P) (where P is the label of the process like `java myPackage.MyClass arg1` for instance) must be associated with the killProcess(P). Hence, using ILR it is possible to express that, for example, in a start procedure containing a startProcess(P) instruction, there must be a killProcess(P) instruction in the stop procedure. Consequently the combination of PNR and ILR ensures that every deployment actions can be cancelled completely without undesirable side effects.

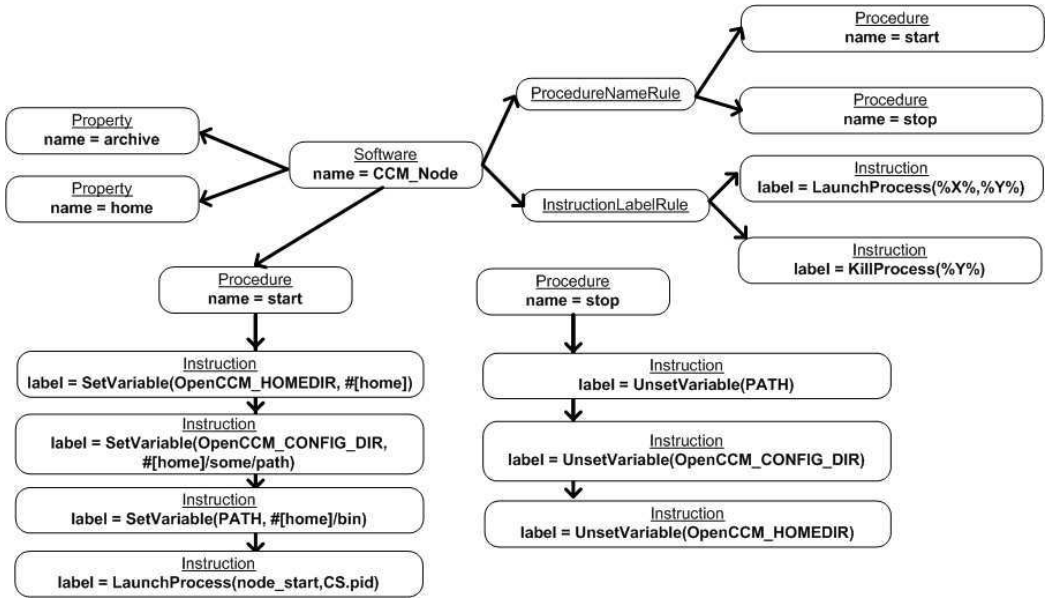

Fig. 5. Instance of a software description : A CORBA Component Server

In Figure 5 is represented an instance of Software representing the CCM component server (discussed in Section 2). Using introspection of this software, it is possible to check, for each action of the install process, and regarding the validation rules, to see if the invert instruction (or procedure) is present. In this case we can see that the `ProcedureNameRule` defined for the CCM server is respected since the software has a procedure start and also a procedure stop. Nevertheless, the `InstructionLabelRule` is not respected. There is, in the start procedure, an instruction labeled `LaunchProcess(%X%,%Y%)` (where `%X%` is the `node_start` command and `%Y%` is `CS.pid`). There should be, in the stop procedure, an instruction labeled `KillProcess(CS.pid)`. Then this software definition is not valid.

Validation of Autonomic Policies — Autonomic behaviour is expressed using rules in DACAR. Then, the problem of interactions between these rules is primordial and must be handled. For that, we introduce a last concept which is called the *Intention Rule* (InR) to achieve static validation of autonomic behaviour. This concept allows to associate a specific Event to an Action. This way the administrator expresses its intention: What change is expected to be performed when executing an action. Using this association of intention, it is possible to compute the sequence of rule triggers according to changes occuring at runtime. Figure 6 describes how we can analyse this sequence, in order to detect cycles, which is one of the most important harmful interaction between policies. From the list of changes likely to occur in the system, it is possible to get the different actions that compose the execution part of the policy. Then, thanks to the Intention Rule, it is possible to compute which events will then be produced due to the execution of these

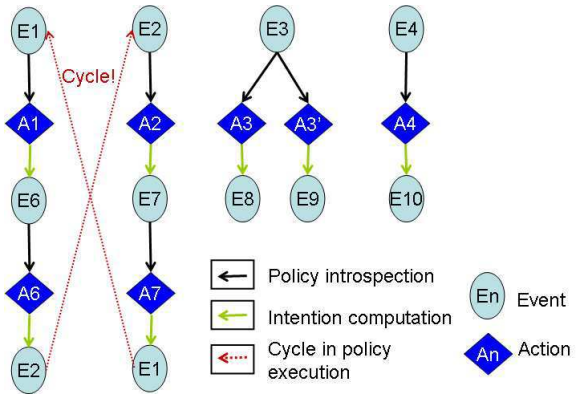

Fig. 6. An exemple of cycle detection thanks to the Intention concept

actions. Using this technique it is also possible to detect *Shared Trigger Interactions* which are detected using a graph as can be seen on Figure 6 when two vertices with different labels emerge from the same event.

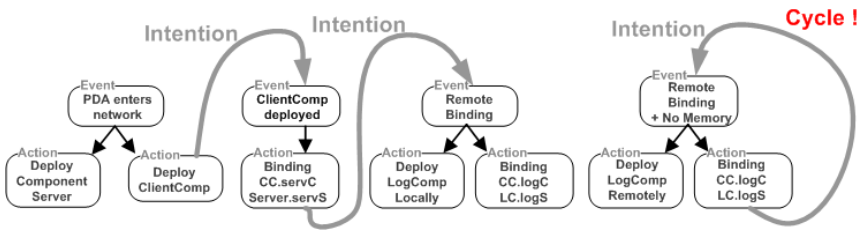

Fig. 7. Analysis of the rule intentions in an example

Figure 7 represents the autonomic policies of the component-based application enounced in Section 2: Using Intention Rules, it is possible to detect that the R4 rule is cycling.

4 Related Work

Jade proposes a component-based framework to build control loops to administrate J2EE applications on clusters [7]. The target platform and the application are modeled using Fractal components [8], in order to provide management interfaces. This allows the administrator to dynamically reconfigure the application architecture. Jade allows the architectures to be reconfigured according to infrastructure context changes. The Jasmine project [3], which strongly relies on Jade, offers an additional design layer to implement autonomic policies expressed using JBoss Rules [4]. This offers a convenient way to express rules, although no verification of the interaction between these rules are possible in contrast with DACAR.

The Rainbow framework [9] also implements a control loop to manage elements across the systems. It defines adaptation strategies using *invariants*, which are reconfiguration scripts executed in response to events. The use of invariants makes the policies in Rainbow monolithic, on the contrary of our approach. Consequently, this disadvantage leads to impossibility to detect interactions, and no verifications about the

[3] `http://jasmine.objectweb.org`
[4] `http://www.jboss.com/products/rules`

global behaviour of the system can be brought. The invariants are programs where only the syntax and the types employed can be statically verified. In addition of these two verifications, DACAR offers a behaviour validation which is crucial for long-life systems. Finally J2EEML is a modeling environment to implement autonomic EJB applications with QoS requirements [10]. These requirements are expressed using the graphical modeling environment and are then woven onto the components of the applications. Then, specific adaptation code is generated to make the EJB components able to be reconfigured according to the defined QoS requirements. This approach generates autonomic behaviour code from the defined model of QoS requirements, which leads to difficulties in introspection of the code, to validate the autonomic behaviour of the system as it is implemented in DACAR. Moreover, this approach seems to be specific to the EJB business components, and also specific to reconfigurations driven by QoS requirements: Reconfigurations due to fluctuations in an ODDE are impossible.

5 Conclusion and Future Work

In this paper, we have presented DACAR supporting model-based framework for autonomic heterogenous distributed software systems in ODDE. DACAR realizes the concepts of autonomic computing. By extending the DACAR metamodel with the adequate concepts we achieve a behaviour validation of autonomic policies. In this paper, two properties are ensured: The deployment and undeployment of Software are symetric, which means that no-side effects occurs when performing these two tasks. The second property is that autonomic policies are running safely, which means that no fatal interactions such as cycles are possible in the global autonomic behaviour. Indeed autonomic policies are classically expressed using programs or scripts, and no behavioural verification is possible, despite existing and well-identified problems in autonomic systems such as the feature interactions. DACAR allows the administrators to validate their whole software deployment thanks to two kinds of validation, the first one validates the correctness of the deployment procedures of the system, and the second one introspects autonomic policies to detect interactions between them. In this paper, two properties are ensured: The deployment and undeployment of Software are symetric, which means that no-side effects occurs when performing these two tasks. The second property is that autonomic policies are running safely, which means that no fatal interactions such as cycles are possible in the global autonomic behaviour.

A prototype of the DACAR metamodel has been developped using the KERMETA [11] metamodeling environment. Validation of several software as they are defined in our DeployWare framework has been successfully experimented. We are also driving experiments using Kermeta and DeployWare to provide an efficient, scalable and validated deployment framework for autonomic software systems. Our future work will mainly consist in consolidating autonomic deployment procedures validation in order to make autonomic deployment really effective and trusted. Another interesting work could be to find mechanisms to automatically infer Intention Rule from the specification of autonomic policies.

References

1. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Technical report, IBM Thomas J. Watson (2003) Published by the IEEE Computer Society.
2. Reiff-Marganiec, S., Turner, K.J.: Feature Interaction in Policies. Computer Networks 45 (2004) 569—584 Department of Computing Science and Mathematics, University of Stirling, United Kingdom.
3. Dubus, J., Merle, P.: Autonomous Deployment and Reconfiguration of Component-based Applications in Open Distributed Environments . In: Proceedings of the 8th International OTM Symposium on Distributed Objects and Applications (DOA'06). Volume 4277 of Lecture Notes in Computer Science, Montpellier, France, Springer-Verlag (2006) 26–27
4. Dubus, J., Merle, P.: Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In: Proceedings of the Models Workshop on Models@Runtime. Volume 4364 of Lecture Notes in Computer Science, Genova, Italia, Springer-Verlag (2006) 242—252

5. Object Management Group: Deployment and Configuration of Distributed Component-based Applications Specification. Available Specification, Version 4.0 formal/06-04-02 (2006)
6. Flissi, A., Merle, P.: A Generic Deployment Framework for Grid Computing and Distributed Applications . In: Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06). Volume 4279 of Lecture Notes in Computer Science, Montpellier, France, Springer-Verlag (2006) 1402–1411
7. Bouchenak, S., Palma, N.D., Hagimont, D., Taton, C.: Autonomic Management of Clustered Applications. In: IEEE International Conference on Cluster Computing, Barcelona, Spain, IEEE (2006)
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11-12) (2006) 1257–1284
9. Garlan, D., Cheng, S.W., Huang, A.C.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer **37** (2004) 46–54 2004.
10. White, J., Schmidt, D.C., Gokhale, A.: Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study. Volume 3713/2005. (2005) 601—615
11. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-Languages. In: Proceedings of MODELS/UML'2005. (2005) 264–278 Montego Bay, Jamaica.